DHBW
Duale Hochschule
Baden-Württemberg
Mosbach

Dokumentation der Projektarbeit

Website zum Vergleich der Spritpreise in einem angegebenen Radius

von

Florian Lohrum (6272128)

Rico Kursidem (5451998)

Frederik Maifarth (2551954)

2021

Inhaltsverzeichnis

Einleitung

Grundidee

Ziel unseres Web Services ist es, einem Benutzer die Möglichkeit zu geben, einfach und schnell, aktuelle Spritpreise in der Nähe zu vergleichen. Dabei soll der Standort vom Nutzer selbst, oder über den Browser direkt angefragt werden. Hiernach kann der Benutzer sich entscheiden welche Spritsorte er präferiert und in welchem Umkreis man die Tankstellen heraussuchen soll. Nach dem alle notwendigen Parameter eingegeben wurden, wird eine Liste der Tankstellen angezeigt. Wenn man nun auf eine Tankstelle drückt, werden noch zusätzliche Informationen, wie der Tankstellenname, die Tankstellenmarke, aktuell geöffnet, sowie alle anderen Spritpreise der spezifischen Tankstelle angezeigt.

Organisation

Projektablauf

Das Projekt begann am 18.01.2022 mit der ersten Vorlesung im Fach: Web-Services. In dieser Vorlesung wurden die Rahmenbedingungen des Projektes geklärt und daraufhin fanden sich verschiedene Teams zusammen. Nach der Teamfindungsphase starteten wir mit einem Brainstorming-Termin. Dort wurden verschiedene Ideen zusammengetragen und schlussendlich sich für das aktuelle Thema entschieden.

In den darauffolgenden Wochen wurden dann die einzelnen Komponenten in Zusammenarbeit entwickelt.

Aufgabenbereiche

Die Folgende Abbildung stellt die Aufgabenbereiche der Mitglieder noch einmal übersichtlich dar.

Name	Aufgabenbereich
Florian Lohrum (6272128)	- WebUI - Performance-Tests - Dokumentation
Frederik Maifarth (2551954)	- BPMN - Dokumentation
Rico Kursidem (5451998)	- Back-End - Dokumentation

Beachten sollte man jedoch bei der Aufgabenverteilung, dass wir als Team sehr verknüpft gearbeitet haben. Dies hat zur Folge, dass jeder in jedem Bereich tätig war, um somit einen maximalen Lernerfolg zu erzielen.

BPMN-Modell

Überblick über die Umsetzung

Der Ablauf der Daten erfolgt folgendermaßen. Zuerst gibt ein Nutzer die Daten (Standort, Radius und Sprit Art) über die Web-Oberfläche ein. Das Front-End übermittelt die Daten an das Backend. Dieses wiederrum greift auf verschiedene APIs zu. Zum einen besorgt es sich den Längen- und Breitengrad des im Front-End angegebenen Standorts über die „WeatherAPI“. Nachdem diese die Koordinaten an das Backend zurückgegeben hat, gibt es die Koordinaten weiter an die „TankerkönigAPI“. Diese ermittelt anhand der Längen- und Breitengrade (Longitude/Latitude) alle umliegenden Tankstellen. Die Daten, welche die „TankerkönigAPI“ an das Backend zurückgibt, setzt sich zusammen aus dem Hersteller der Tankstelle, dem Spritpreis, der Öffnungszeiten sowie der Adresse der Tankstelle. Nachdem das Backend alles empfangen hat, schreibt es die Daten in eine JSON-Datei und gibt diese dem Front-End.

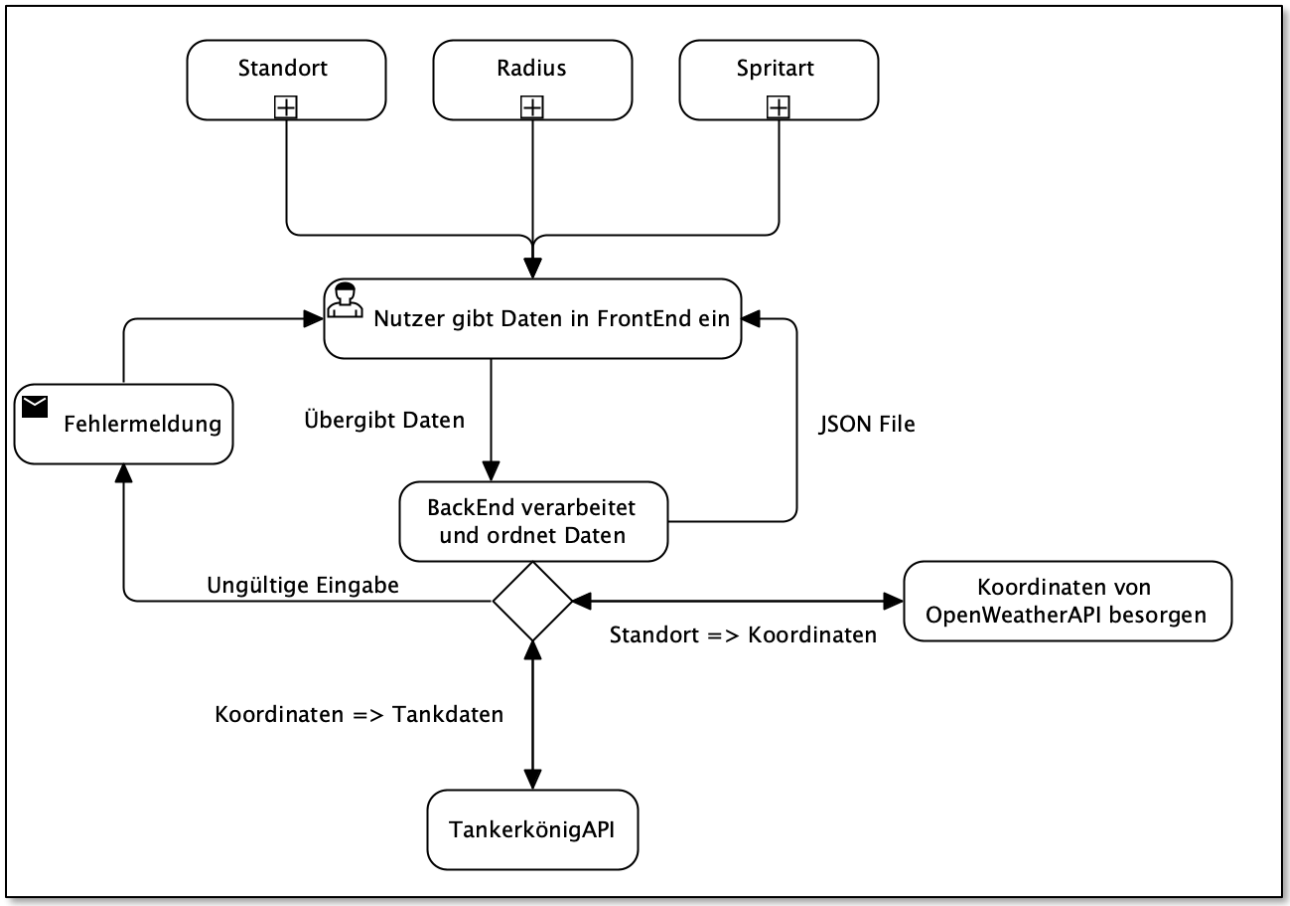

Abbildung 1 -BPMN-Modell der Webanwendung

OpenAPI

Im folgendem ist die OpenAPI Beschreibung, die mit Swagger erstellt wurde. Ebenfalls finden Sie diese als yaml-Datei im Verzeichnis des Projektes.

```
swagger: "2.0"
info:
  description: "An small API to get gasstations around your Area. It uses
OpenWeather API and TankerKoenig API to convert your location in gasstations
around you."
  version: "4.2.0"
  title: "Tank-o-Mator"
  contact:
    email: "tankiboy420@tankmail.de"

tags:
- name: "default"
  description: "A way to test the functionality of the API."
- name: "tank-o-mator"
  description: "All requests to our own API"

paths:
  /:
    get:
      tags:
      - "default"
      responses:
        "200":
          description: "It works"
  /api/gasStation:
    get:
      tags:
      - "tank-o-mator"
      parameters:
      - name: "location"
        in: "query"
        type: "string"
        description: "The location"
        required: true
      - name: "radius"
        in: "query"
        type: "string"
        description: "The radius"
        required: true
      responses:
        "200":
          description: "Everything is fine"
        "404":
```

```
    description: "No gas station found"
  "500":
    description: "internal Server Error. Explanation in description."
```

Implementierung

Performance-Test

Die Performance-Tests wurden mit dem Tool: Postman durchgeführt. Dabei wurde zwei Szenarien getestet. Das erste Szenario ist eine Anfrage in Mosbach mit dem Radius von 10 km. Das zweite Szenario ist eine Anfrage in Heidelberg mit dem Radius von 5 km. Die jeweiligen Szenarios wurden mit 100 Iterationen durchgeführt und ein Delay von 100ms nach jeder Iteration.

Folgender Code wurde für das Testen benutzt:

```
var time = pm.response.responseTime;
pm.test(`Status code is correct, correct location and succesfull POST request - Res
ponse time: ${time}ms`, function () {
    pm.response.to.have.status(200);
    var data = pm.response.json();
    pm.expect(JSON.stringify(data[0]['location'])).to.eql('"Heidelberg"');
    setTimeout(function(){}, [100]);
});
```

Der Test überprüft, ob der richtige Statuscode vorherrscht. Ebenfalls überprüft er ob der Standort richtig übernommen wurde. Ebenfalls wird noch die responsetime ausgegeben um ein Vergleich der einzelnen Anfragen zu ermöglichen.

Folgend sind die Ergebnisse der ersten Iteration:

Abbildung 2 Performance-Test Iteration 1

Folgend sind die Ergebnisse der zweiten Iteration:

Abbildung 3 Performance-Test Iteration 2

Wie zu erwarten unterscheiden sich die Zeiten stark. Dies liegt daran, dass wir die Abfragen in einer Datenbank gecached (zwischengespeichert) haben. Dadurch muss keine neue Anfrage mehr gemacht werden und somit kann man schneller auf die Daten zugreifen, die zuvor schon angefragt wurden. Dabei werden die Ergebnisse der Anfrage für 5 Minuten aufgehoben und danach verworfen.

Durch mehrfache Ausführung der Tests ist folgendes aufgefallen:

- Die Anfrage ohne Caching dauert im Schnitt: 160ms
- Die Anfrage mit Caching dauert im Schnitt: 10ms

Natürlich ist dabei zu beachten, dass es immer mal wieder zu größeren Unterschieden kommen kann. Dies könnte vor allem an der Auslastung, der Webservern, der öffentlichen APIs liegen.

Back-End

Das Backend besteht aus einer Flask API welche in einem seperaten Dockercontainer läuft. Dieser Container ist von Außerhalb des Systems erreichbar.

Als Programmiersprache wird Python genutzt da sich die Gestaltung einer Flask API sehr einfach umsetzen lässt. Außerdem gibt es für komplexe Problemstellung einfach zu implementierende Bibliotheken, welche die Entwicklungszeit verkürzen und Zuverlässiger machen.

Ablauf eines API-Aufrufes

Wird eine Get-Request mit der URL */api/gasStation* und den richtigen Parametern aufgerufen, werden die Parameter an das Python Skript “gas_station.py” übergeben.

```
jsonify(gasStationMain(request.args['location'],request.args['radius'])
```

In diesem Skript werden dann beide Anfragen an die Externen Dateien formuliert und durchgeführt. Nach jeder Abfrage der externen APIs werden die Statuscode und die restliche Response-JSON überprüft und bei Fehlern ein Fehlercode an das Frontend gesendet.

Erfolgten alle Anfragen so werden die erhaltenen Daten Formuliert und in einer Response JSON formuliert und geordnet. Die Response JSON wird daraufhin an das Frontend zurückgegeben.

```
{
'status': 400,
'location': Berlin
'Stations': [ {Daten aller Tankstellen in einer Liste }]
}
```

Docker

Die Anwendung ist in zwei Docker Container aufgebrochen. Das Frontend und die API. Für das Frontend wurde ein `nginx:alpine` Image genutzt und das Backend arbeitet mit einem `python: 3` Image als Grundlage. Um die Bibliotheken einzubinden wird eine *requirements.txt* erstellt welche automatisch durch das Dockerfile beim Erstellen des Containers geladen wird und alle Bibliotheken in dieser Datei werden Installiert. Dies wird durch die folgenden Zeilen im Dockerfile erzeugt.

```
COPY requirements.txt /app

RUN pip install --upgrade pip

RUN pip install -r requirements.txt
```

Außerdem wird in diesem Dockerfile der Port 5000 des Containers geöffnet und die Python Applikation wird ausgeführt.

```
EXPOSE 5000

CMD [ "python", "api.py" ]
```

Um beide Container gemeinsam in derselben Umgebung zu starten wird eine Docker-Compose Datei benötigt. Diese erstellt aus beiden Dockerfiles die Container und leitet die Ports der Hostmaschine an die Container weiter. Außerdem wird sichergestellt das das Frontend nur existieren kann, wenn das Backend geladen hat da der Web-App Container nur funktioniert, wenn die API zur Gewinnung der Daten zur Verfügung steht.

```
version: '3.9'

services:

  api:

    build: ./BackendAPI

    hostname: api

    ports:

      - "5000:1273"


  webapp:

    build: ./Frontend

    ports:

      - "80:1337"

    depends_on:

      - api
```

Front-End

Die WebUI wurde mit Hilfe von HTML, CSS & JavaScript erstellt. Dabei wurde mit dem CSS-Framework Bootstrap, in der Version 5.1 gearbeitet. Durch Bootstrap ist es möglich schnell und einfach eine Weboberfläche zu erstellen, die einerseits gut aussieht und andererseits responsive ist. Dies ermöglicht es uns, dass der Benutzer die Website auf verschiedene Endgeräte benutzen kann. Bei der Implementierung wurde insbesondere darauf geachtet, dass die Website dem Nutzer eine einfache und intuitive Steuerung bietet. Der Nutzer soll nicht von vielen Funktionen und Unterseiten abgelenkt sein, sondern soll sich auf den wesentlichen Service konzentrieren. Dies hat den weiteren Vorteil, dass das Aufrufen und die Spritpreissuche einfach und schnell vonstattengeht und somit der Nutzer in kürzester Zeit die wichtigsten Informationen erhält. Ebenso wurde die komplette WebUI im Darkmode erstellt. Dies bedeutet, dass man auf sehr helle Farben verzichtete und zu dunkeln Farben griff um somit ein angenehmeres Bild für das Auge zu schaffen.

Neben Bootstrap wurde noch Font Awesome benutzt. Dies ist eine schriftartbasierte Icon-Sammlung. Das heißt die Icons können auch wie eine normale Schriftart behandelt werden und mittels CSS zum Beispiel eingefärbt und frei skaliert werden. In dem Webprojekt wurde die momentan aktuellste Version (6.0.0) genutzt. Mittels Font Awesome ist es nun auch möglich animierte Icons einzubauen und so wurde beispielsweise die Ladeanzeige realisiert.

Grundsätzlicher Aufbau

Die Website lässt sich in 3 Teilbereiche gliedern, die Startseite, eine Seite für die Verweise und eine Seite für die Anzeige der gefundenen Tankstellen. Dabei ist zu beachten, dass die Startseite und die Seite für die Anzeige auf eine Website basiert, nur die Verweisseite lässt sich über eine andere URL erreichen. Die Verweisseite ist simpel gehalten und soll einen übersichtlichen Einblick in die genutzten Frameworks und APIs liefern. Die Startseite besteht im Wesentlichen aus einer Navigationsbar, einem einführenden Text und der 3 Eingabefelder, die die Auswahl für die Suche darstellen. Wenn man, nach Eingabe der nötigen Werte, nun auf die Schaltfläche „suchen“ drückt, werden die verschiedenen Tankstellen die gefunden wurde, als eine Tabelle von Akkordeon-Elementen angezeigt. Zu allererst werden die wichtigsten Informationen der jeweils gefundene Tankstelle angezeigt, darunter der Name, die gesuchte Sprit Art, die Zeit der zuletzt geschehenen Aktualisierung, sowie der Preis der gesuchten Spritsorte. Mit einem Klick auf die jeweilige Tankstelle ist es möglich noch weitere Informationen über diese sich anzeigenzulassen. Darunter Zählt: Die Marke, ob die Tankstelle momentan geöffnet oder geschlossen ist, die genaue Adresse der Tankstelle, sowie die Preise der anderen Spritsorten.

Die angezeigten Tankstellen sind nach der Distanz sortiert. Die Tankstelle mit dem günstigsten Spritpreis erkennt man daran, dass der Preis grün, anstatt weiß eingefärbt ist. Des Weiteren wurde darauf geachtet, dass der Nutzer bei falschen Angaben darauf

hingewiesen wird, denn Standort zu überprüfen, falls zwar der Standort korrekt ist, aber keine Tankstelle sich in der Nähe befindet, wird das dem Nutzer ebenfalls mitgeteilt.

Besonderheiten der Implementierung

Wie oben schon erwähnt wurde die Website mit einfachem HTML, CSS und JavaScript implementiert. Dabei werden die wichtigsten Container schon zu beginn geladen, diese haben aber das Attribut: „display: none;“. Dies bedeutet, dass einzelne Elemente noch gar nicht angezeigt werden. Erst wenn verschiedene Ereignisse eintreten werden diese angezeigt. Dies ist über JavaScript realisiert, in dem man den verschiedenen Elementen Klassen hinzufügt, beziehungsweise löscht:

```
<script>
      //Funktion zum Anzeigen der Inputs im Reihenformat (Nach Tankstellen suche)
      function showInputrow() {
          var divShow = document.getElementById("inputAsRow");
          var divInvis = document.getElementById("inputAsColumn")
          var accordion = document.getElementById("accordion");
          var titleText = document.getElementById("titleText");
          document.getElementById("ortRow").value =
document.getElementById("ortColumn").value;
          divShow.classList.remove("d-none");
          divShow.classList.add("d-flex");
          divInvis.classList.remove("d-flex");
          divInvis.classList.add("d-none");
          accordion.classList.remove("d-none");
          accordion.classList.add("d-flex");
          titleText.classList.add("d-none");

          createAccordion(document.getElementById("spritartColumn").value)
      }
```

In diesem Codebeispiel werden zuerst die Elemente via ID gesucht. Hiernach werden per class.List.remove bzw. class.List.add einzelne Klassen hinzugefügt bzw. gelöscht. Dies hat zur Folge, dass bestimmte Elemente eingeblendet werden und andere wiederum ausgeblendet.

Ebenso werden auf der Website alle Inputs, die dieselbe Funktion haben synchronisiert, dies geschieht ebenfalls mit JavaScript. Im obigen Codesnippet wird beispielweise das Element mit der ID: „ortRow“ und das Element mit der ID: „ortColumn“ synchronisiert, in dem die Werte gleichgesetzt werden.

Für die dynamische Erstellung der Anzeige der Spritpreise wird die Funktion createAccordion(spritArt) genutzt. In dieser findet die Anfrage an unsere selbstgeschriebene API statt um die Notwendigen Werte zu bekommen. Nach dem dies geschehen ist, wird die Funktion createData(data, spritArt) aufgerufen. Diese Initialisiert zu Beginn die notwendigen Werte und mittels einer „for-Schleife“ wird dann jedes Akkordeonfeld einzeln erstellt und mit dem Befehl: accordion.innerHTML += „data (Aus Übersichtlichkeitsgründen gekürzt)“

dem schon existierenden Container hinzugefügt. Bei einem erneuten Aufruf dieser Funktion wird zuerst die innerHTML des Akkordeoncontainers gelöscht und wieder neu erstellt.

Aufruf der eigenen API:

```
  //API Anfrage nach den Daten
  let request = new XMLHttpRequest();
  request.open('GET','http://localhost:5000/api/gasStation?location='+document
.getElementById("ortRow").value+'&radius='+document.getElementById("radiusRow"
).value);
  request.send();
  request.onload = () => {
    if(request.status == 200){
      console.log(JSON.parse(request.response)[0]['status']);

      if(JSON.parse(request.response)[0]['status'] == "error"){
        document.getElementById("errorText").innerHTML ="";
        document.getElementById("errorText").innerHTML +="\n
"+JSON.parse(request.response)[0]['description']
        document.getElementById("noGasStation").classList.remove("d-none");
        document.getElementById("loading").classList.add("d-none");
      }
      else{
        createData(JSON.parse(request.response), spritArt);
      }

    }
    else{
      console.log(`error ${request.status} ${request.statusText}`);
    }
  }
}
```